Combining Different Implementations of Types in a Program[1]

Xavier Franch
franch@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Technical University of Catalunya (UPC)
Pau Gargallo 5, 08028 Barcelona, Catalunya (Spain)
FAX: 34-3-4017014. Phone: 34-3-4017006

Abstract

This paper presents a methodology to support the existence of multiple type implementations when programming with a modular, abstract-data-type oriented language. Multiple implementations arise from the need to separate specifications and implementations in programs: objects are declared of a type according to their expected behaviour and later they are attached to an implementation according to their efficiency requirements; so, different objects of the same type may be attached to different implementations. Implementations may be selected by means of *ad hoc* language constructs in the appropriate contexts; another language construct, the abstraction function, allows implicit switching between implementations during execution. As an additional advantage, the abstraction function supports prototyping of incomplete programs provided that there exists an operational specification for non-implemented types.

Keyword Codes: D.1.4, D.2.5, D.3.3

Keywords: Sequential Programming; Testing and Debugging; Language Constructs and Features

1. Introduction

Development of programs in a modular, abstract-data-type (ADT for short) oriented framework follows the information hiding principle in such a way that algorithms depend only on the specification of the types they use. To obtain a final program, a single implementation is chosen for every type and so their corresponding objects[2] are bound just to that implementation. This is the case when writing programs in Modula-2, Ada, Pascal, C and other Algol-like languages, where every type should present a single module to define its operations and a different module for each implementation.

There are contexts where it would be useful to allow different objects of a type to be represented with different implementations, mainly for efficiency purposes. For instance, there could exist two objects *s* and *t* of type *set*, the first one implemented with hashing to improve individual access, and the other one implemented with an ordered list to support ordered listing. In spite of their different implementations, we claim that the type must be the same for both objects in order to provide the programmer with a high abstraction degree: algorithms may be built knowing just the specification of the type, not its implementation. As a result, it should be possible to have different implementations of the same type in a program.

This fact has a main consequence. As far as the type for *s* and *t* is the same, the expression $s \cap t$ should be supported by the language, but how can it be executed? Algorithms for $\cap$ must

---

[1]This work is partially supported by the spanish research programme PRONTIC under contract TIC92-0667.

[2]Throughout this paper, the term "object" stands for variable, parameter, constant or function result in an imperative program; it is not used with its meaning in the object-oriented framework.

exist in both implementation modules, but probably not for the mixed case; even more, such an algorithm should not exist, because in this case many functions not really attached to a concrete implementation would appear, making programs too implementation-dependent and forcing programmers to be aware of the need to provide those functions when needed. Instead, a mechanism to map an object from one representation to another should be provided in the language.

A language designed to support this methodology, **Merlí**, is presented. ADTs in Merlí are specified in a module that serves as an interface to their users (at least, the name of the types and the exported operations -that is, the **signature** of the type- are listed; optionally, a equational specification may be provided), and they are implemented in other modules; details of the representation are hidden to the users. Merlí modules are called **universes**; they may define new types (probably parameterised), or enrichments of types by adding new operations on some existing ones; depending on their contents, we talk about **specification universes** or **implementation universes**. Objects must be declared of a type and they may be attached to a particular implementation using the appropriate features of the language. In order to support the change from one implementation to another, the abstraction function of the implementation must be provided, the goal of which is to transform any instance of the data structure that implements the type to an expression consisting of a sequence of operations that may produce it; so, to turn from one implementation to another, it suffices to apply the abstraction function to the source representation and then executing the resulting term into the destination representation. As an additional feature, the abstraction function allows programs to be executed even if there exist non-implemented types, provided that they present an operational specification which may be transformed into a set of rewriting rules; then, the specification itself may be considered as an implementation and so the general scheme applies.

2. The Multiparadigm Programming Language *Merlí*

In this section we describe the most relevant features of the programming language Merlí (see [Fra92] and [FBB93] for a complete description). As it has been said, Merlí can be used to write specifications and implementations and this is why it can be classified as a **multiparadigm** programming language.

Merlí specification universes are used to define type signatures and properties. Properties are stated in equational style, the equations being pairs of **terms** (expressions) formed from the corresponding Herbrand universe; equations may be conditional, the conditions being boolean terms. The logic to express equations allows to associate initial semantics (see next section) to specifications [ADJ78] and thus it is possible to use a term-rewriting system to manipulate terms, provided that the set of equations may be transformed into a canonical set of rewriting rules [HuO80].

Special interest is given to erroneous situations; in the way of [ADJ78], a special error-value is (implicitly) introduced in every type under definition. All erroneous terms are identified and grouped inside a special Merlí clause, and the remaining (non-erroneous) equations may be interpreted as error-free. Moreover, error-values propagate: the term *f(..., error, ...)* yields to an error-value of the appropriate type.

On the other hand, implementation universes are written in imperative style, using a notation that resembles Ada-like programming languages (with some additional mechanisms not presented here, see [Fra92]). Each implementation universe must be referred to a specification one (which presents at least the signature of the type) and it may include data structures, functions and procedures. The invariant of the representation and the equality interpretation of the implementation may be defined with the data structure, as well as the abstraction function introduced in section 4.

In both kinds of universes some structuring mechanisms (in the way of [BuG77]) can be included. The basic one is the use of specifications, which makes all the symbols exported by a specification universe available to its users. There is also a parameterisation facility to define generic universes, coupled with a parameter passing mechanism; formal parameters are defined in the so-called **characterisation universes**. Last, there are some constructs to control the scope of symbols in universes: symbols may be renamed, hidden or introduced as private from the very beginning; uses and instantiations can also be public or private.

Finally, we mention that there are no predefined types in the language, except for booleans. Instead, a standard library has been defined containing the usual types and type constructors with the usual notation; so, the user may redefine types by providing new universes for them. We also remark that this library includes ADTs modelling input-output devices; so, there are neither notation nor methodological differences in writing to a printer and writing to a hash table. Files are not included in the library; in fact, there is no way to define files in the language, and the procedure to store data in disk is by classifying objects as persistent (i.e., retaining their value between executions) or not (the default).

As an example, in the next page we give the specification and (part of) an implementation of a parameterised type for (simplified) sets, as well as an example of instantiation for sets of naturals. The formal parameters are the element's type, *elem*, and the maximum size of the set, *val*; *elem* is defined in the characterisation universe *ELEM*, while *val* is defined inside *VAL*; each parameter is classified as a type identifier or an operation and, in the second case, its properties are stated. Some of the language capabilities are: the use of the specification of natural numbers, the (optional) prefixing of symbols with the name of the universe that defines them, the definition of private functions, the specification of erroneous situations, the use of conditional equations, the renaming of symbols, the implementation of ADT operations by functions or procedures indistinctly, and the existence of an equality operator for elements and naturals (which is automatically defined for all existing types, meaning the equational calculus for specifications -see section 5- and the equality interpretation in imperative code).

3. Selecting Implementations for Types

Up to now, an outline of the Merlí programming language has been presented. In this section, it is shown how to select implementations in Merlí modules, and next the rules to determine which implementation is bound to every object in a program are stated.

Let *Uimp* be an implementation universe. There are two ways to select a type implementation inside *Uimp*:

- Universe level: use and instantiation of universes. Let *T* be the name of an used or instantiated universe, let *Timp* be the name of an implementation universe for *T*, and let t_1, ..., t_n be the types introduced by *T* (either by direct declaration or by instantiation). Then, the declarations "**uses** T **implemented with** Timp" or "**instantiates** T (P_1, ..., P_k) **implemented with** Timp" makes all the objects declared of a type t_i inside *Uimp*, $1 \leq i \leq n$, to be implemented with *Timp*.
- Object level: declaration of objects. Let *x* be the name of a variable, parameter or constant declared of a type *t* or else to denote a function result of type *t*, and let *Timp* be the name of an universe that implements type *t*. Then, the declaration "x: t **implemented with** Timp" in those contexts makes *x* to be implemented with *Timp*, regardless of any existing binding done at universe level.

From these language constructs, each object appearing in an universe may be attached to a single implementation. Note that different objects of the same type may in fact be implemented in different ways, according to the former rules. It should be remarked that just a single implementation may appear in the "**implemented with**" clause.

```
universe SET (ELEM,VAL) defines
  uses NAT
  type set
  ops
    empty: → set
    put: set elem → set
    in?: set elem → bool
    empty?: set → bool
    private nelems: set → nat
  errors c: set; v: elem
    [nelems(c) = val ∧ ¬ in?(c,v)] ⇒ put(c,v)
  eqns c: set; v,v1,v2: elem
    put(put(c,v),v) = put(c,v); put(put(c,v1),v2) = put(put(c,v2),v1)
    in?(empty,v) = false; in?(put(c,v1),v2) = (v1 = v2) √ in?(c,v2)
    empty(c) = (nelems(c) = 0)
    nelems(empty) = 0
    [in?(c,v)] ⇒ nelems(put(c,v)) = nelems(c,v)
    [¬ in?(c,v)] ⇒ nelems(put(c,v)) = nelems(c,v) NAT.+ 1
end universe

universe ELEM characterises
   type elem
end universe

universe VAL characterises
   uses NAT
   ops val: → nat
   eqns val>10 = true
end universe
```

Listing 1: specification of sets.

```
universe SET_OF_50_NAT defines
  uses NAT
  instantiates SET (ELEM,VAL) where elem is nat, val is 50
  renames set by set_of_50_nats
end universe
```

Listing 2: example of instantiation of sets.

```
universe SET_IMPL_SEQ(ELEM,VAL) implements SET(ELEM,VAL)
 uses NAT
 type set is
   record
      A: array [0..val-1] of elem; free: nat
 end type
 function empty returns set is
 var c: set end var
   c.free := 0
 returns c
 procedure put (in/out c: set; in v: elem) is
   if ¬ in?(c,v) then
      if c.free = val then error
      else c.A[c.free] := v; c.free := c.free+1
      end if
   end if
 end procedure
 ...
end universe
```

Listing 3: sequential implementation of sets.

This informal description is not complete enough, because rules have been stated just for isolated universes, without taking into account the whole module hierarchy of an application. When considering it, conflicts may appear when an object has more than one associated implementation. Thus, a more accurate description is needed.

First of all, we define several syntactic domains of interest. Let *U* be an universe and let *H* be a module hierarchy. We define the domains:

> τ_U = { *t* / *t* is a known type inside *U* }. That is, τ_U stands for those types for which objects may be declared inside *U*.
>
> π_U = { *t* / *t* is a type exported by *U* }. That is, π_U stands for those types known in all the modules using/instantiating *U*. It holds that $\pi_U \sum \tau_U$.
>
> ω_U = { *x* / *x* is an object whose scope is contained in *Uimp* }. That is, ω_U stands for all the objects in *U* whose implementation has to be determined.
>
> Θ_H = { *V* / *V* is an implementation module attached to any specification one in *H* }. That is, Θ_H stands for all the possible implementations for types and objects in *H*. Θ_H presents a special value, $\perp$, to denote the non-existent implementation; it is used to indicate that a type or an object has no implementation universe attached yet.

Note that if *U* is an implementation universe for a specification *V*, π_U is the same in every other implementation universe for *V*, and it equals the set of types defined in *V* as public, π_V; this set is formed by the types enumerated in the "**type**" clause as public, plus the types exported by the public uses and the public instantiations done in the universe. On the other hand, τ_U is likely to vary among different implementation universes, because different strategies may require different auxiliary types.

Next, we introduce a pair of functions to bind types and objects to their corresponding implementation universes at the syntactical level. Let *Uimp* be an implementation universe. We define:

> T_{Uimp}: $\tau_U \rightarrow \Theta_H$, a function that binds types to the implementation universe used to represent them.
>
> I_{Uimp}: $\omega_U \rightarrow \Theta_H$, a function that binds objects to the implementation universe used to represent them. Determining I_{Uimp} is the goal of this section, because it fixes the implementation of every object.

The definition of both functions is:

- T_{Uimp}:

  R1. For every new type *t* introduced in *Uimp* with a certain representation, "**type** t = … **end type**", the implementation is the universe itself, $T_{Uimp}(t)$ ::= *Uimp*.

  R2. For every used or instantiated universe *V* with implementation *Vimp* appearing in *Uimp*, "**uses** V **implemented with** Vimp" or "**instantiates** V(P_1, ..., P_k) **implemented with** Vimp", it holds that:

  $$\forall t: t \in \pi_V: T_{Uimp}(t) ::= T_{Vimp}(t)$$

  This rule implements the propagation of implementations through the hierarchy of modules. Propagation takes place in a bottom-up manner, following module relationships. Eventually, as we will see later, two different paths in the hierarchy may yield to potential implementation clashes. One may think that this propagation rule is too powerful because implementations in very low level modules may become inadequate in higher level ones; however, as far as implementations should be totally determined to end the application, propagation may be seen as the default, being the programmer able to change this default with the appropriate constructs.

R3. For every used or instantiated universe *V* without associated implementation appearing in *Uimp*, "**uses** V" or "**instantiates** V(P_1, ..., P_k)", such that R2 does not fix an implementation for their types, it holds that:

$$\forall t: t \in \pi_V: T_{Uimp}(t) ::= \perp$$

That is, the use or instantiation of a specification leaves the selection open to the object level, unless an implementation is selected using R2 following another path in the hierarchy. It should be noted that the use or instantiation of a specification prevents the selection of implementations of any type used by this specification.

Note that these rules do not depend on whether the types are public or not in *Uimp*; this attribute affects only to the set π_V.

- I_{Uimp}:

  For every constant *x* in the universe, for every procedure or function parameter *x*, for every local variable *x* in a procedure or function and for every function result denoted by *x*, being *t* the type of *x* and *V* the specification universe defining *t*, the implementation for *x* is:

  S1. The implementation universe for the type *t*, if no specification is stated, $I_{Uimp}(x) ::= T_{Uimp}(t)$. So, the selected implementation for the type acts as a default value for their objects.

  S2. The implementation universe *Vimp* stated in its declaration, if it exists (using the "**implemented with**" clause), $I_{Uimp}(x) ::= Vimp$. Note that this definition may override the default value associated to the type *t*, $T_{Uimp}(t)$.

To sum up, the function I_{Uimp} fixes the implementation of every object in the whole hierarchy, as it was required to; if no implementation is selected, the value $\perp$ is associated to the object.

We show an example application of these rules. In figure 1 we present a hierarchy of four specification universes, defining each one a single type; note that an implicit "**uses**" relationship exists from *A* to *D*, due to transitivity of uses. Obviously, the "**uses**" relationship may not pick out implementations for modules at the specification level. In figure 2 we complete the hierarchy providing various implementation modules. Some of them refer to particular implementations of the specifications they use; also, one of them introduces an auxiliary type to implement its type of interest. As a result, many types have a default implementation for their objects in some modules of the hierarchy, as shown in table 1 (where an 'x' stands for unknown types in the module; the sets π_U and τ_U of every implementation universe are included too). Note that module *Cimp1* provides a default implementation for all its types, as also trivially do all the implementations for *D*. Note the propagation of the implementation *Dimp1* for *d* when using *Cimp1* from *Aimp2*, applying R2.

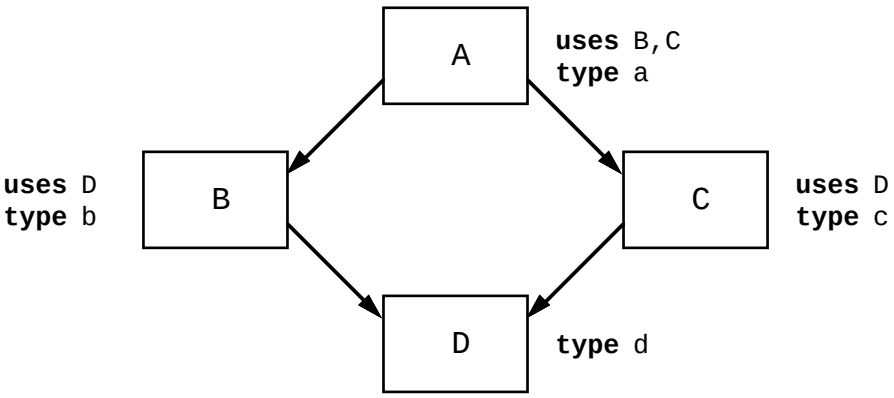

Figure 1: a hierarchy of modules.

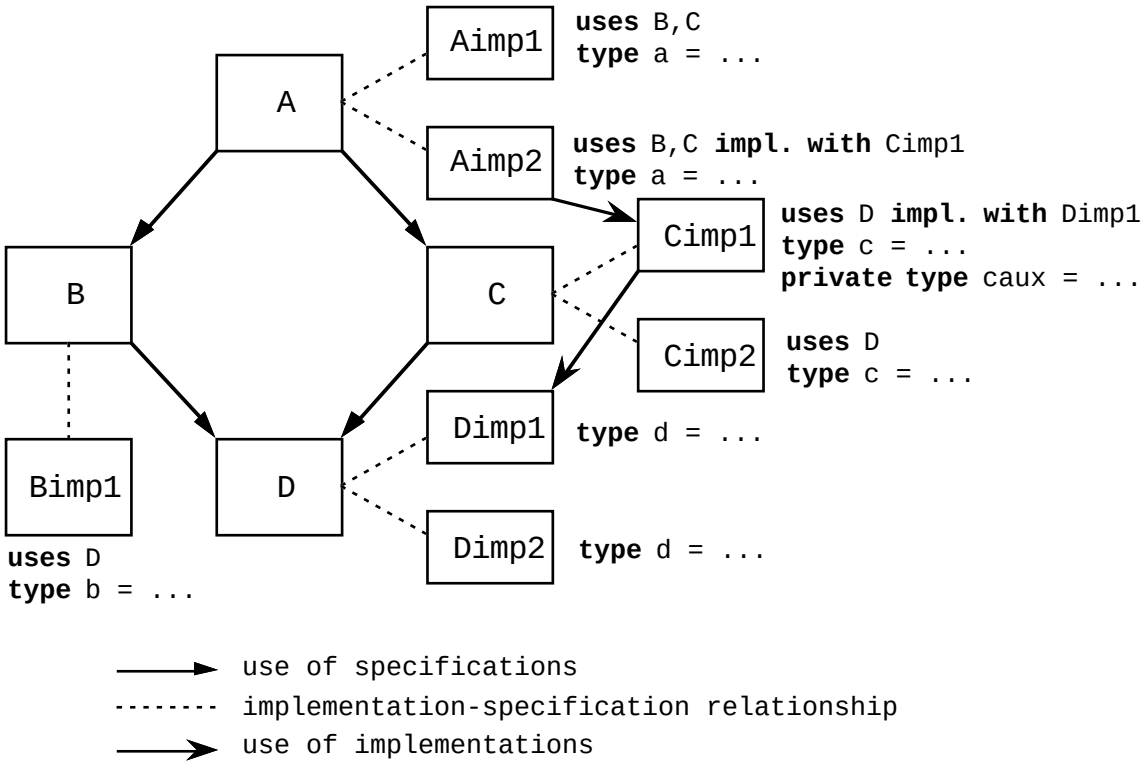

Figure 2: a hierarchy of modules with implementations.

	T_U(a)	T_U(b)	T_U(c)	T_U(caux)	T_U(d)	τ_U	π_U
Aimp1	Aimp1	⊥	⊥	x	⊥	a b c d	a b c d
Aimp2	Aimp2	⊥	Cimp1	x	Dimp1	a b c d	a b c d
Bimp1	x	Bimp1	x	x	⊥	b d	b d
Cimp1	x	x	Cimp1	Cimp1	Dimp1	c caux d	c d
Cimp2	x	x	Cimp2	x	⊥	c d	c d
Dimp1	x	x	x	x	Dimp1	d	d
Dimp2	x	x	x	x	Dimp2	d	d

Table 1: selection of implementations in the modules of figure 2.

In order to execute the application, it is necessary first to choose among *Aimp1* and *Aimp2* which one will act as implementation for the main specification. For instance, in the case of *Aimp2*, it just remains to fix an implementation for *b* objects at universe level (obviously *Bimp1*; one can expect the execution environment to make these kind of selections automatically, possibly giving a warning to the user) or else to select the implementation for *b* objects one by one; also, different implementations could be given to individual objects of any type but *a*, according to rule S2. During prototyping, however, selection of implementations could be delayed (see section 5).

There is one question left. Assume that, whatever the implementation for *A* is, the chosen implementation for *D* must be *Dimp2*. This can be easily stated by making explicit the use of *D* in *A* with the clause "**uses** D **implemented with** Dimp2". This variation works correctly in *Aimp1*, but it does not in *Aimp2*, because a implementation clash turns out: this use of *Dimp2* collides with the former implementation selected with propagation, *Dimp1*. So, rules must be reformulated to deal with this case, the goal being that direct uses or instances of implementations must beat propagation. First, we define a sub domain of π_U, ν_U, by grouping all the new types introduced in an universe:

ν_U = { *t* / *t* is a type introduced in *U* directly or by an instance }.

Using the specification universe *U* with a given implementation *Uimp* makes T_{Uimp} the implementation for all the types in ν_U , if any, regardless of other implementations for *U* determined in lower levels of the hierarchy.

Next, we reformulate the rules dividing R2 into four new ones, two of them being integrity constraints (R2.iii and R2.iv). Also, R1 and R3 are more formally stated. Let *Uimp* be an implementation module, let X be the set of specification universes that are directly used or instantiated in *Uimp* with an associated implementation, let Y be the set of specification universes that are directly used or instantiated in *Uimp* without an associated implementation, and for all *V* in X, let *Vimp* be the associated implementation for *V*.

R1. For every new type *t* introduced in *Uimp* with a certain representation, the implementation is the universe itself:

$$\forall t: t \in \rho_{Uimp}: T_{Uimp}(t) ::= Uimp$$

where ρ_{Uimp} stands for types introduced directly in *Uimp*.

R2.i. New types defined and implemented in directly-used implementations override type propagation:

$$\forall V: V \in X:$$
$$\forall t: t \in \nu_V \wedge T_{Vimp}(t) \neq \perp: T_{Uimp}(t) ::= T_{Vimp}(t).$$

$T_{Vimp}(t)$ will be *Vimp* itself in the case of directly-implemented types, or the implementation given for the generic universe when the type is obtained *via* instantiation.

R2.ii. Types not defined as new and/or not implemented in directly-used implementations do follow type propagation:

$$\forall V: V \in X:$$
$$\forall t: (t \notin \nu_V \surd T_{Vimp}(t) = \perp) \wedge (\neg \exists W: W \in X: t \in \nu_W \wedge T_{Wimp}(t) \neq \perp):$$
$$T_{Uimp}(t) ::= T_{Vimp}(t).$$

Note that the value of T_{Uimp} is the same as before, but rules differ in the quantification domain. This difference is fundamental, because it determines which universe must be selected in case of conflict.

R2.iii. Types defined as new and implemented in directly-used implementations should not have more than one implementation selected:

$$\forall V,W: V \in X \wedge W \in X:$$
$$\forall t: (t \in \nu_V \wedge T_{Vimp}(t) \neq \perp) \wedge (t \in \nu_W \wedge T_{Wimp}(t) \neq \perp):$$
$$T_{Vimp}(t) = T_{Wimp}(t).$$

This conflict can only appear from two identical instances of parameterised universes, but with different implementations. The clash may be easily avoided by giving the resulting types different names.

R2.iv. Type propagation should not select more than one implementation:

$$\forall V,W: V \in X \wedge W \in X:$$
$$\forall t: (t \notin \nu_V \surd T_{Vimp}(t) = \perp) \wedge (t \notin \nu_W \surd T_{Wimp}(t) = \perp):$$
$$T_{Vimp}(t) = \perp \surd T_{Wimp}(t) = \perp \surd T_{Vimp}(t) = T_{Wimp}(t).$$

This case is an unavoidable clash. No decision (for instance, selecting the shortest path of "**uses**" relationship) seems reasonable. So, the user should solve the conflict either by giving an explicit use to apply R2.i or else by removing enough implementation uses in the hierarchy to avoid the conflict (which will not be advisable in the general case).

R3. Types defined in used or instantiated universes without implementation attached do not have any implementation, unless rule R2 fixes one of them:

$$\forall V: V \in Y:$$
$$\forall t: t \in \pi_V \wedge (\neg \exists W: W \in X: t \in \nu_W \wedge T_{Wimp}(t) \neq \perp):$$
$$T_{Uimp}(t) ::= \perp.$$

To end up this section, it is worth extending our approach to the semantical level in a way such that coexistence of implementations gets a mathematical meaning. This may be easily done in the algebraic framework, where heterogeneous algebras are the models for data types. Informally speaking, an **heterogeneous algebra** (algebra for short) consists of a collection of data domains (called **carrier sets**) and a set of functions over them. Algebras are associated to both signatures and specifications: let Σ be a signature, $\Sigma = (S, OP)$, being S the set of types and OP the set of operations, and let SP be a specification for Σ, $SP = (\Sigma, E)$ or also $SP = (S, OP, E)$, being E the set of equations; an algebra A is called a Σ-algebra if the mapping between types and operations in Σ, and carrier sets and functions in A is one-to-one, and it is called a SP-algebra if it is a Σ-algebra and it satisfies also the equations in SP.

In order to be able to execute specifications, it is necessary to take **initial algebras** as models for abstract data types [ADJ78]; initial algebras are mainly characterised by the "no confusion" rule: two terms are said to be equal if their syntactical equivalence may be proved from the equations; else, they are said to be different. The mechanism for proving this equivalence is called **equational calculus** [EhM85] and it is based in applying equations of the type in the terms as far as possible; equational calculus is the basis of the term-rewriting systems used to execute specifications [HuO80].

In our framework, not only specifications have an initial algebra (the quotient term algebra, see section 5), but also do implementations. This algebra may be inferred if we build an algebraic specification of Merlí, in the way of [BWP87]; then, a Merlí data type implementation for a specification $V = (S, OP, E)$, contained in an universe *Vimp*, has an algebra $A_{Vimp} = (S_A, OP_A)$ defined as follows:

- There is a carrier set t_A in S_A for every type t in S. Each carrier set may be determined from:
    1. The model of the components of the type representation.
    2. The concrete type representation.
- There is a function f_A: $t1_A..tn_A \rightarrow t_A$ in OP_A for every operation f: $t1...tn \rightarrow t$ in OP. The behaviour of each function (which must satisfy the equations in E) may be determined from:
    3. The model of the objects appearing in the function.
    4. The semantics of the language constructs (fixed from their own specification).
    5. The concrete type function implementation using the language constructs.

Note that 4 is the algebraic specification of Merlí, while 2 and 5 are the Merlí code; so, it is necessary to fix 1 and 3 to determine completely the model of an implementation. The only task left is to bind any reference to an object with the appropriate semantics in the algebra for this object, which will have been calculated in turn with the same rules above. To do this, it suffices to convert T_v and I_v from syntactical functions to semantic ones, associating to every type or object the algebra of the associated implementation.

Let *Uimp* be an implementation universe. We define:

T_{Uimp}: $\tau_{Uimp} \rightarrow Alg_H$, a function that binds types to the model of their implementation.

I_{Uimp}: $\omega_{Uimp} \rightarrow Alg_H$, a function that binds objects to the model of their implementation.

with Alg_H = { A_{Vimp} / A_{Vimp} is the algebra associated to any implementation module *Vimp* for a specification module *V* in a hierarchy of modules *H* }

The definition of both functions is:

- T_{Uimp}:

  P1. For every type *t* whose implementation universe is determined, the model for *t* is the algebra of that implementation:

  $$T_{Uimp}(t) = Vimp \wedge Vimp \neq \perp \Rightarrow T_{Uimp}(t) ::= A_{Vimp}$$

  P2. For every type *t* whose implementation universe is not determined, the model for *t* is the empty algebra Ø (no carrier sets and no operations):

  $$T_{Uimp}(t) = \perp \Rightarrow T_{Uimp}(t) ::= Ø$$

- I_{Uimp}:

  Q1. For every object *x* whose implementation universe is determined, the model for *x* is the algebra of that implementation:

  $$I_{Uimp}(x) = Vimp \wedge Vimp \neq \perp \Rightarrow I_{Uimp}(x) ::= A_{Vimp}$$

  Q2. For every object *x* whose implementation universe is not determined, the model for *x* is the empty algebra Ø:

  $$I_{Uimp}(x) = \perp \Rightarrow I_{Uimp}(x) ::= Ø$$

As a result, program semantics may be calculated by composition of their participant models (using homomorphisms between algebras), using I_{Uimp} and Merlí semantics.

4. Implementation Switching

Once the implementation of every object is fixed, a question immediately arises: is it possible for two objects of the same type but different implementations to be used in the same context? This section addresses this particular point and introduces a new language feature, the abstraction function, which is necessary to support this behavioural equivalence.

Given a type *t* specified in an universe *T* and implemented with universes *Timp1* and *Timp2*, given an operation *f*: *t t* → *t* defined in *T* and given two objects *x* and *y* of type *t*, *x* implemented with *Timp1* and *y* implemented with *Timp2*, the former question may be summarised as: is the call *f*(*x*, *y*) executable or not? Obviously, both *Timp1* and *Timp2* are required to have their own implementation for *f*, built on their particular representation of type *t*, and they should be not aware of whether other implementations for *t* do exist. So, other alternatives to support the execution of these "mixed" expressions are considered:

- To write a collection of implementations for *f*, one for every "interesting" (or even possible) combination of its parameter implementations.

- To explicitly define conversion functions between any pair of implementations for *t*.
- To define a function to convert objects from any implementation for *t* to a common domain, and vice versa.

The first two methods suffer from a lack of modularity (where are those functions to be put in?) and they are not well suited for program maintenance (construction of new implementations require modification of older ones). So, we choose the third option, and then it is necessary to fix the common domain.

The simplest solution is to define a pair of functions, the **abstraction function** and the **representation function**. The first one [Hoa72] maps an object to an equivalent term at the specification level, while the second one performs the inverse mapping. Note that the abstraction function must be explicitly provided for every implementation of a type, while the representation function is just the execution of the operations contained in the term into the chosen implementation.

More formally, let Σ_T to denote the **term algebra** for the signature Σ [EhM85], that is, the Σ-algebra $\Sigma_T = (S_\Sigma, OP_\Sigma)$ defined as:

- The carrier sets are composed by the terms built from the signature Σ. There is a different carrier set σ_t for every type *t* in Σ.
- The operations are the rules for composition of terms. That is, for every operation f: $t1 \ldots tn \rightarrow t$ in Σ there is an operation f_Σ: $\sigma_{t1} \ldots \sigma_{tn} \rightarrow \sigma_t$ such that $f_\Sigma(x_1, \ldots, x_n)$ is defined as the term $f(x_1, \ldots, x_n)$.

To sum up, Σ_T is the set of terms over Σ plus the rules to form those terms. Then, we define:

$abs_{V,t}$: $A_V \rightarrow \Sigma_T$, as the abstraction function for a type $t \in \Sigma$ implemented in a module *V*.

$repr_{V,t}$: $\Sigma_T \rightarrow A_V$, as the representation function for a type $t \in \Sigma$ implemented in a module *V*.

The important point is that, given an object *x* of type *t*, its implementation *V* may be changed by first applying $abs_{V,t}$ over it and then $repr_{W,t}$, being *W* the destination implementation.

Given a public type *t*, its abstraction function can be provided in two ways, using both notations of the multiparadigm language:

- Equational style. Expressed by means of conditional equations, it is usually defined as a kind of recursive function that builds the resulting term applying repeatedly operations of the type, usually with the help of several quantifiers offered by the language.
- Imperative style. Mainly used when the function is difficult to express in equational style. It uses a variable of a predefined generic type *term* to store the result; operations on *term* are those ones from the signature of *t*, and so terms from this signature may be built.

Equational style is not expected to be as efficient as imperative one, due to its execution with a term-rewriting system; however, this assumption may fail because the use of recursive functions is likely to permit the application of lazy evaluation techniques in some contexts (see next section). In both styles, there could exist auxiliary functions[3].

Listings 4 and 5 present two examples of abstraction functions. The first one is the abstraction function for sequential sets as defined in listing 3, given in equational style; recursion is done over the free position in the array, being the empty array the trivial case. The second one is an abstraction function for an adjacency matrix implementation of directed graphs (with operations

---

[3]Even more, in an equational abstraction function there could exist an auxiliary imperative function, and vice versa. See next section for execution details in this case.

to obtain the empty graph and to add an edge to the graph); in this case, an imperative program consisting of two nested loops is the simplest solution.

```
abs (s: set) is
   abs(<A,0>) = empty
   [n > 0] ⇒ abs(<A,n>) = put(abs(<A,n-1>),A[n-1])
```

Listing 4: an abstraction function for sequential sets.

```
type graph = array [vertex,vertex] of bool end type
abs (g: graph) is
var t: term end var
   t := empty
   for all v in vertex do
       for all w in vertex do
            if g[v,w] then t := add(t,v,w) end if
       end for all
   end for all
returns t
```

Listing 5: an abstraction function for directed graphs implemented with adjacency matrix.

Let *t* be a type specified in an universe *T* and let *Vimp* and *Wimp* be two different implementation universes for *T*. There are four situations that require to apply the abstraction function:

T1. Assignment rule. Given two objects *x* and *y* of type *t* introduced in an implementation universe *Uimp* and implemented with *Vimp* and *Wimp* respectively, $I_{Uimp}(x)$ = *Vimp* and $I_{Uimp}(y)$ = *Wimp*, the assignment *x* := *y* requires to transform *y* from *Wimp* to *Vimp*:

$$x := y \rightarrow x := repr_{Vimp,t}(abs_{Wimp,t}(y))$$

T2. Parameter rule. Given an object *x* of type *t* introduced in an implementation universe *Uimp* and implemented with *Vimp*, $I_{Uimp}(x)$ = *Vimp*, and given a function or procedure *f* implemented in an universe *Fimp* with a formal parameter *y* of type *t* declared to be implemented with *Wimp*, $I_{Fimp}(y)$ = *Wimp*, calling *f*(…*x*...) with *x* as the actual parameter bound to *y* requires:

T2.i. To transform *x* from *Vimp* to *Wimp* before executing *f*, if *y* is an "**in**" parameter:

$$f(...x...) \rightarrow f(...repr_{Wimp,t}(abs_{Vimp,t}(x))...)$$

T2.ii. To transform *y* from *Wimp* to *Vimp* after executing *f*, if *y* is an "**out**" parameter:

$$f(...x...) \rightarrow f(...aux...);\ x := repr_{Vimp,t}(abs_{Wimp,t}(aux))$$

being *aux* of type *t* implemented with *Wimp*.

T2.iii. To apply both T2.i and T2.ii, if *y* is an "**in/out**" parameter.

$$f(...x...) \rightarrow aux := repr_{Wimp,t}(abs_{Vimp,t}(x));$$
$$f(...aux...);\ x := repr_{Vimp,t}(abs_{Wimp,t}(aux))$$

being *aux* of type *t* implemented with *Wimp*.

T3. Function result rule. Given a function *f* implemented in an universe *Fimp* with a result *x* of type *t* implemented with *Vimp*, $I_{Fimp}(x) = Vimp$, using the function in any context α demanding a value of type *t* implemented with *Wimp* requires to transform *x* from *Vimp* to *Wimp*:

$$\alpha(f(...)) \rightarrow \alpha(repr_{Wimp,t}(abs_{Vimp,t}(f(...))))$$

T4. Compatibility rule. Given two objects *x* and *y* of type *t* introduced in an implementation universe *Uimp* and implemented with *Vimp* and *Wimp* respectively, $I_{Uimp}(x) = Vimp$ and $I_{Uimp}(y) = Wimp$, and given a function *f* appearing in an universe *Fimp* that implements *T* with two formal parameters *p* and *q* of type *t*, calling *f*(...*x*...*y*...) with *x* and *y* as the actual parameters bound to *p* and *q* requires:

T4.i. To transform *y* from *Wimp* to *Vimp* if *Fimp* = *Vimp*, or to transform *x* from *Vimp* to *Wimp* if *Fimp* = *Wimp*. This will be the usual case where an implementation universe acts as default, and an individual object has been chosen with a different one.

$$Fimp = Vimp \Rightarrow f(...x...y...) \rightarrow f(...x...repr_{Vimp,t}(abs_{Wimp,t}(y))...)$$

$$Fimp = Wimp \Rightarrow f(...x...y...) \rightarrow f(...repr_{Wimp,t}(abs_{Vimp,t}(x))...y...)$$

T4.ii. To transform *x* from *Vimp* to *Fimp* and to transform *y* from *Wimp* to *Fimp*, if $Fimp \neq Vimp$, $Fimp \neq Wimp$ and $Fimp \neq \perp$. Normally, it will happen that *Vimp* = *Wimp*, being the default implementation universe, and *Fimp* will be an universe where *f* has any interesting property.

$$Fimp \neq Vimp \wedge Fimp \neq Wimp \wedge Fimp \neq \perp \Rightarrow$$

$$f(...x...y...) \rightarrow f(...repr_{Fimp,t}(abs_{Vimp,t}(x))...repr_{Fimp,t}(abs_{Wimp,t}(y))...)$$

T4.iii. To transform *y* from *Wimp* to *Vimp* or to transform *x* from *Vimp* to *Wimp* indistinctly and to call *f* with the corresponding implementation, if *Fimp* is not determined. *Fimp* will be not determined when no implementation universe has been chosen globally, but locally to objects. It should be noted that the concrete election may depend on the context; for instance, if *f* returns a value of type *t* and the context demands this value to be implemented with *Vimp*, it should be *w* the object to be transformed.

$$Fimp = \perp \Rightarrow$$

$$f(...x...y...) \rightarrow f(...repr_{Wimp,t}(abs_{Vimp,t}(x))...y...) \ \surd$$

$$f(...x...y...) \rightarrow f(...x...repr_{Vimp,t}(abs_{Wimp,t}(y))...)$$

In any case, the transformation will be done applying the rule T2 with respect to the parameter mode.

An example will show how these rules may be applied during execution. Given the hierarchy of figures 1 and 2, suppose that *C* defines three operations *f1*: $\rightarrow c$, *f2*: $c \rightarrow c$ and *f3*: $c\ c \rightarrow c$ and that these operations are implemented in the module *Cimp1* as functions for *f1* and *f2* and as a procedure for *f3*, **function** *f1* **returns** *c*, **function** *f2* (*p*: *c*) **returns** *c*, **procedure** *f3* (**in/out** *p*: *c*; **in** *q*: *c*). Also, imagine that the program will be executed using the implementation *Aimp2*, in which there exist three variables of type *c*, declared as **var** *x*, *y*: *c*; *z*: *c* **implemented with** *Cimp2*. Provided that the value for $T_{Aimp2}(c)$ is *Cimp1*, so are $I_{Aimp2}(x)$ and $I_{Aimp2}(y)$, while $I_{Aimp2}(z)$ is clearly *Cimp2*. With this scenario, in figure 3 we present a collection of assignments and function/procedure calls, showing the rule or rules applied in every case (if any).

x := y	no rule applied
x := z	rule T1; means x := $repr_{Cimp1,c}(abs_{Cimp2,c}(z))$
x := f1	no rule applied
z := f1	rule T3; means: z := $repr_{Cimp2,c}(abs_{Cimp1,c}(f1))$
x := f2(y)	no rule applied
x := f2(z)	rule T2.i; means: x := $f2(repr_{Cimp1,c}(abs_{Cimp2,c}(z)))$
z := f2(z)	rules T2.i and T3; means: z := $repr_{Cimp2,c}(abs_{Cimp1,c}(f2(repr_{Cimp1,c}(abs_{Cimp2,c}(z)))))$
z := CIMP2.f2(z)	no rule applied
f3(x,y)	no rule applied
f3(x,z)	rule T4.i, using T2.i for z; means: $f3(x, repr_{Cimp1,c}(abs_{Cimp2,c}(z)))$
f3(z,x)	rule T4.i, using T2.iii for z; being aux an object of type c implemented with $Cimp1$, means: aux := $repr_{Cimp1,c}(abs_{Cimp2,c}(z))$; $f3(aux, x)$; z := $repr_{Cimp2,c}(abs_{Cimp1,c}(aux))$
CIMP2.f3(x,y)	T4.ii, using T2.iii for x and T2.i for y; being aux an object of type c implemented with $Cimp2$, means: aux := $repr_{Cimp2,c}(abs_{Cimp1,c}(x))$; $f3(aux, repr_{Cimp2,c}(abs_{Cimp1,c}(y)))$; x := $repr_{Cimp1,c}(abs_{Cimp2,c}(aux))$

Figure 3: switching of implementations.

5. Execution of Incomplete Programs

In the preceding sections, coexistence of different implementations for types has been studied in order to obtain complete programs, that is, programs such that every object has a bound implementation. Next, we are going to see how the language constructs already presented may be used to support the execution of incomplete programs, i.e. programs which may contain non-implemented types, and/or objects with no associated implementation. Execution of incomplete programs is interesting in the context of program development through prototyping [BaG88], because the choice of implementation for types and objects may be postponed until the entire behaviour of the module hierarchy has been proved correct. We propose next a methodology to perform this "proof" through testing, by means of the construction of a sequence of executable prototypes.

Prototyping in Merlí arises naturally from the fact that the language is a multiparadigm one. So, development of programs may start from stating the specification of the modules in the hierarchy; if those specifications are operational, execution is possible using a term-rewriting system. At this first step of prototyping, execution consists of proving properties on types, and then it is possible to correct any detected misbehaviour before implementing. Once the hierarchy

is specified, modules may be implemented one by one, being able to prototype the application in any intermediate step provided that the abstraction function for the implementations does exist[4]. The details of the execution process are given later.

To sum up, we are interested in providing a framework where intermediate prototypes, those with one or more non-implemented modules and one or more objects with no implementation attached, may be executed. As a previous step, for those non-implemented objects whose type is implemented in one or more universes, it seems reasonable to bind any of these universes to the object when prototyping, and so the problem is reduced to the case of having non-implemented specification modules.

The approach we present here is based on combining a term-rewriting system for specifications and an interpreter to execute imperative code. To facilitate their communication, a single internal representation of programs is shared by both tools, which is a tree resulting from a classical syntax-directed editor. Object values in the tree will be terms for non-implemented objects and a data structure for implemented ones; both kinds of values will be intertwined in the tree at any degree. The switch from one execution tool to the other has to be decided when applying an operation to some parameters and depends on the state of the universes that define them:

- If the operation has already been implemented, we distinguish two situations:
  - If all the parameters are implemented objects, which will be the case if a representation for their type exists, the interpreter proceeds in a normal way.
  - Otherwise, the interpreter may also act, because parameters (that are bound to types defined inside other universes) may be manipulated only by means of their own operations.
- If the operation is not implemented but specified, we also distinguish many different situations:
  - If all the parameters are specified objects, the term-rewriting system proceeds in a normal way.
  - If some parameters are implemented objects, the abstraction function is applied to them and the term-rewriting system may proceed too.
  - If the operation is neither implemented nor specified in an operational way, prototyping fails.

In fact, the execution system is more sophisticated than shown here (see [BBF88]). For instance, depending on the concrete appearance of the left-hand side of the rules, the abstraction function may not be needed, although perhaps the equality interpretation will; if needed, perhaps it will not be necessary to obtain the entire term but only a part of it, just to be able to apply a few rewriting rules. For instance, let's borrow a simple example from [EhM85, p. 58] where a *line-editor* is specified using the type *string*, which we suppose implemented in an universe *Istring*. We focus on the specification of an operation:

$$delete: line\text{-}editor \rightarrow line\text{-}editor$$

to delete the current character in the line, made up of the *string* operations[5]:

---

[4]This description of prototyping in Merlí has been simplified, because it is not the aim of the paper; see [Fra92] and [FrB93] for more details. Actually, the whole process is very flexible and so not all modules need to be specified, nor the implementation rate must be one module at time; even more, prototyping may be possible involving not totally implemented modules. As a result, determining the sequence of prototypes is up to the developer.

[5]These are the so-called **generating operations** in the initial semantics framework; that is, the minimal set of operations whose combination allows to generate terms to represent all the ADT values.

empty: → *string*, and

add-left: *char string* → *string*

and the *line-editor* operation[5]:

line: *string string* → *line-editor*,

which composes two strings to form a line, being the cursor on the first character of the right string. Two versions are presented, which are correct into the initial semantics approach:

- There are no restrictions over the equations: being *r* and *s* of type *string* and *c* a character:

  delete(line(r, empty)) = line(r, empty) -- cursor at the end of line

  delete(line(r, add-left(c, s)) = line(r, s)

  In this case, the evaluation of an expression like *delete(line(x, y))* in a module *U*, being *x* and *y* of type *string* such that $I_U(x) = I_U(y) = Istring$, forces the abstraction function for *string* to be applied over *y*, because the term represented by *y* is needed to choose the right equation to apply. However, if the abstraction function is in equational style, just an abstraction step is needed to distinguish if *y* is the empty string or not, resulting in a kind of lazy evaluation that improves execution speed.

- Only operations of *line-editor* may be referenced in the left-hand side of the equations, with no restriction over the right-hand ones; then, no abstraction is needed:

  [s = empty] ⇒ delete(line(r, s)) = line(r, empty)

  [s ≠ empty] ⇒ delete(line(r, s)) = line(r, delete-left(s))

  In this case, conditions will be evaluated by the interpreter using the equality interpretation, which should then exist. Once an equation is selected, the term-rewriting system may proceed. The existence of an operation *delete-left*: *string* → *string*, which is the inverse for *add-left*, is needed on strings; if not, this option is not applicable.

Last, we study the variations on the semantic in this prototyping framework. Let $V = (\Sigma, E)$ be a specification module, $\Sigma = (S, OP)$. We denote by Γ_V the **quotient term algebra** over *V* [EhM85], that is the *V*-algebra $\Gamma_V = (S_\Gamma, OP_\Gamma)$, where:

- There is a different carrier set $\gamma_t \in S_\Gamma$ for every type *t* in *S*. The components of γ_t are the terms in Σ_T grouped in equivalence classes, using as equivalence relation the equational calculus; in other words, two terms are in the same class if their syntactical equivalence may be derived from the equations in *E*.

- There is an operation $f_\Gamma: \gamma_{t1}...\gamma_{tk} \rightarrow \gamma_t \in OP_\Gamma$ for every function *f: t1...tk* → *t* in *V*, such that $f_\Gamma(x_1, ..., x_n)$ is defined as the equivalence class that contains the term $f(x_1, ..., x_n)$.

Then, we redefine T_{Uimp} and I_{Uimp} as follows:

- Their domain is the same as before, $dom(T_{Uimp}) = \tau_{Uimp}$ and $dom(I_{Uimp}) = \omega_{Uimp}$.
- Their rank is enlarged with the quotient term algebras of the specifications in the hierarchy, $Alg_H = \{ A_{Vimp} / A_{Vimp}$ is the model associated to any implementation module *Vimp* for a specification module *V* in a hierarchy of modules *H* $\} \cup \{ \Gamma_V / \Gamma_V$ is the quotient term algebra associated to any specification module *V* in the hierarchy of modules *H* $\}$.

- The rules which define to T_{Uimp} and I_{Uimp} are:

  P1. $T_{Uimp}(t) = Vimp \wedge Vimp \neq \perp \Rightarrow T_{Uimp}(t) = A_{Vimp}$

  P2. $T_{Uimp}(t) = \perp \Rightarrow T_{Uimp}(t) = \Gamma_V$

  Q1. $I_{Uimp}(x) = Vimp \wedge Vimp \neq \perp \Rightarrow I_{Uimp}(x) = A_{Vimp}$

  Q2. $I_{Uimp}(x) = \perp \Rightarrow I_{Uimp}(x) = \Gamma_V$

  being *V* the specification universe defining type *t* and being *x* of type *t*.

6. Comparison with other Work

The idea of explicitly binding implementations with program objects in the ADT framework is not new (see [Ben87]); however, up to know we do not know of any language with the characteristics described in this paper. So, comparison is done with the most closely related approach: object-orientedness. Nevertheless, the difference between Merlí and the classical imperative framework should be first stressed.

Standard imperative programming languages are not well suited for building really implementation-independent programs, because of the need to bind a type to a single implementation. For instance, given two objects *s* and *t* of the same type *t* but designed to have different implementation Imp_x and Imp_y it is necessary to introduce two different types t_x and t_y with the same operations as *t* but linked to Imp_x and Imp_y, respectively; this gives rise to unnecessary complexity of software construction, maintenance, reusability and comprehension.

Object-orientedness provides many interesting features with respect to implementations of types. For instance, we may consider the hierarchy built in the Eiffel library to define classes for data structures [Mey90]. Classes are arranged using the inheritance mechanism following two rules:

- Class *A* inherits from class *B* if the data structure defined in *A* is a specialisation of the one defined in *B*. So, there exists a class *container* to store elements of any type, offering operations for putting elements in *container* objects and also to test for membership; *container* heirs (that is, *container* specialisations) add many operations on it, as *table* does by associating a key to retrieve elements.
- Class *A* inherits from class *B* if the data structure defined in *B* is implemented by the data structure defined in *A*. This kind of relationship replaces the implementation binding between modules that exists in the algebraic framework. So, the classes *hash_table* and *indexable* inherit from *table* because both of them provide different strategies for implementing tables. In this case, operations on both classes are the same.

With this scenario, polymorphism and dynamic binding provide the programmer with a great flexibility. For instance, given the objects *x* from class *table*, *y* from class *hash_table* and *z* from class *indexable*, the assignments *y := x* and *z := x* are valid and make the heirs to take the form of their parent; this behaviour is useful, because programs may be written using the class *table* as an ADT yielding to implementation-independent programs and, once a representation is chosen for *table* objects, they may be converted to it by means of polymorphism; even more, two different objects of class table may be bound to two different heir classes allowing thus multiple implementations to coexist. However, switching of implementations is not possible: the chain *x := y*, *z := x* fails because there is an assignment from a heir to its parent, which is forbidden because it could force the parent class to exhibit properties that are specific of the heir; also, the direct assignment *z := y* is not supported because there is no way to transform implicitly from one class to another. In both cases, it is necessary to explicitly define methods

to transform objects from one class to another, disturbing class design and class utilisation (conversion is not implicit); furthermore, addition of new heirs requires to study the convenience of writing those methods in them and also in existing classes and, if this is the case, to write them with the corresponding programming extra time.

7. Conclusions and Future Work

A framework for combining different implementation of objects has been presented. Combination of different implementations is useful mainly for efficiency reasons, because different objects of the same type may have different efficiency requirements; also, abstraction (programmers really can forget implementation issues when using ADTs), readability (type names are the same regardless of their implementation), reusability (existing modules with their objects implemented in a particular way may be integrated in an application), management (no duplication of modules is needed by implementation differences) and maintenance (changing the requirements on an object is likely to imply changing only its implementation) are improved with multiple implementations.

Our approach has been completed with the ability to switch between implementations, and then objects of the same type may be used in the same contexts regardless of their implementations. Also, as a beneficial side-effect, prototyping of incomplete programs by combination of specifications and implementations has been shown to naturally fit in the multiple implementations framework, provided that specifications are operational.

Some possible drawbacks of this strategy should be mentioned. On the one hand, programmers are obliged to design abstraction functions, which requires an additional coding effort. However, our experience shows that this extra time is not really a drawback in the general case, because abstraction function design forces programmers to think carefully about the correctness of their implementations, and this may help to discover errors or lacks in type representations; in particular, complicated abstraction functions are likely to show incorrect type design. It is worth remarking that the non-existence of a type abstraction function does not prevent the entire scheme to work out: it just prevents multiple implementations for this particular type to interact.

Another potential drawback is execution efficiency. Since it may be necessary to transform from an implementation to a term and then to a different implementation, it is necessary extra space to store the term and extra time to build it and to execute it.

Future work is related to the following issues:

- Implementation: at this moment, the architecture of the system is very simple and it does not cover the special language features presented here. Currently, the environment (called **Excalibur**) contains just a parser and the execution subsystem formed by a preliminary version of a term-rewriting system for equations and an interpreter for imperative programs; both tools are being constructed *ad hoc* using Eiffel (a preliminary version using the Axis term-rewriting system [Axis88] exists but it has been rejected due to the lack of interface and debugging facilities) and they are working independently for the moment. We need to connect them and to support switching of implementations; also, we should incorporate lazy evaluation techniques to improve efficiency.
- Language constructs for combining implementations: we are currently investigating the possibility to bind objects not just to an implementation but to a set of them, provided that often more than one implementation fulfils efficiency requirements on a program; in this sense, it may be useful to classify implementations into categories (linked, sequential, ordered, ...) to be used instead of names of concrete implementations. Finally, the adequacy of an *ad hoc* language construct to require explicit transformation of implementations at any place of the program should be studied .

- Object-orientedness and paralelism: we are currently defining object-orientedness constructs in Merlí; however, this is not the case of paralel aspects, because we prefer to fit the whole scheme in the sequential framework before any extension into the paralel world is done.
- Scope of problem resolution: up to now, the type of problems we have solved are ADT-oriented, with very clear modules and interfaces; most of them have turned up from the academic world (university courses and the writing of a book about data structures [Fra94]). Real industrial cases should be now addressed.

8. References

[ADJ78] J.A. Goguen, J.W. Thatcher, E.G. Wagner. "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", in *Current Trends in Programming Methodology*, Vol. IV, Prentice-Hall, 1978.

[Axis88] P. Arnold. "The User Reference Manual for the Axis System", Technical Report HPL-ISC-TR-88-034, 1988.

[BaG88] R. Balzer (chairm.), R.P. Gabriel (ed.), F. Belz, R. Dewar, D. Fisher, J. Guttag, P. Hudak, M. Wand. "Draft Report on Requirements for a Common Prototyping System", SIGPLAN Notices, 24(3), 1988.

[Ben87] J. Bentley. “Abstract Data Types”, from *Programming Pearls*, Communications ACM, 30(4), Apr. 87.

[BBF88] P. Botella, X. Burgués, X. Franch. "Combining the Imperative and the Equational Programming Paradigms in the Excalibur System", in *Proceedings of the Workshop on Software Engineering and its Applications*, Toulouse (France), 1988.

[BuG77] R.M. Burstall, J.A. Goguen. "Putting Theories together to make Specifications", in *Proceedings of 5th International Joint Conference on Artificial Intelligence*, Cambridge M.A., 1977.

[BWP87] M. Broy, M. Wirsing, P. Pepper. "On the Algebraic Definition of Programming Languages", ACM TOPLAS, 9(1), Jan. 87.

[EhM85] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 1*, EATCS Monographs on Computer Science, 1985.

[FBB93] X. Franch, X. Burgués, P. Botella. "Report on the Programming Language Merlí", Technical Report LSI-93-T (written in catalan), Catalunya Technical University (UPC).

[Fra92] X. Franch. "Supporting Prototyping with a Multiparadigm Language", Research Report LSI-92-12-R, Catalunya Technical University (UPC).

[Fra94] X. Franch. *Data Structures: Specification, Design and Implementation*, ed. Catalunya Technical University (written in spanish), 1994.

[FrB93] X. Franch, X. Burgués. "A case study on prototyping with specifications", *in Proceedings of ERCIM Workshop on Development and Transformation of Programs*, Nancy (France), 1993.

[Hoa72] C.A.R. Hoare. "Proof of Correctness of Data Representations", in *Programming Methodology*, Springer-Verlag, 1972.

[HuO80] G. Huet, D.C. Oppen. "Equations and rewrite rules: a survey", in *Formal Language Theory: Perspectives and Open Problems*, Academic Press, 1980.

[Mey90] B. Meyer. “Lessons from the design of the Eiffel libraries”, Communications ACM, 33(9), Sept. 90.